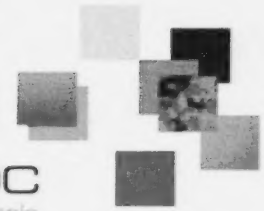
Defence Research and Development Canada | Recherche et développement pour la défense Canada

CAN UNCLASSIFIED



DRDC | RDDC
technologysciencetechnologie

Wordlist password cracking using John the Ripper

A tutorial and lessons learned for heterogeneous clusters

Richard Carbone
DRDC – Valcartier Research Centre

Digital Forensics Magazine
Issue 34
Pages 64–69

Date of Publication from External Publisher: February 2018

Defence Research and Development Canada
External Literature (P)
DRDC-RDDC-2018-P025
March 2018

Canada

IMPORTANT INFORMATIVE STATEMENTS

This document was reviewed for Controlled Goods by Defence Research and Development Canada (DRDC) using the Schedule to the *Defence Production Act*.

Disclaimer: This document is not published by the Editorial Office of Defence Research and Development Canada, an agency of the Department of National Defence of Canada but is to be catalogued in the Canadian Defence Information System (CANDIS), the national repository for Defence S&T documents. Her Majesty the Queen in Right of Canada (Department of National Defence) makes no representations or warranties, expressed or implied, of any kind whatsoever, and assumes no liability for the accuracy, reliability, completeness, currency or usefulness of any information, product, process or material included in this document. Nothing in this document should be interpreted as an endorsement for the specific use of any tool, technique or process examined in it. Any reliance on, or use of, any information, product, process or material included in this document is at the sole risk of the person so using it or relying on it. Canada does not assume any liability in respect of any damages or losses arising out of or in connection with the use of, or reliance on, any information, product, process or material included in this document.

© Her Majesty the Queen in Right of Canada (Department of National Defence), 2018

© Sa Majesté la Reine en droit du Canada (Ministère de la Défense nationale), 2018

The Quarterly Magazine for Digital Forensics Practitioners

Issue 34 • February 2018

DIGITAL FORENSICS Magazine

Forensics Europe Expo
Intelligence & Investigations for the Internet of Things
DFM Sponsored Seminar

Drone Forensics

How the growing use of Unmanned Aerial Vehicles creates new forensic challenges for investigators

PLUS

- Inconsistent Tool Performance
- Faster Searching for Illegal Content
- Data Destruction on current hard disks
- **From the Lab:** Device Forensics in the IoT

9 772042 061004 34
Issue 34 / £14.99
TR Media

Wordlist Password Cracking Using 'John the Ripper'

Richard Carbone provides a tutorial and lessons learned for heterogeneous clusters.

Recently, I needed to recover a passphrase from a retired Linux system to mount a TrueCrypt (TC) volume on that very same disk. At the same time, I had forgotten the passphrase to a more recent EncFS volume. I wrote this article to share my experiences using the John the Ripper (JtR) password cracker atop a small heterogeneous cluster.

Context

Password cracking is a bit like the ugly child of digital forensics; everyone has seen one but almost no one admits to having one.

That said access to commercial password cracking solutions (hardware and software) is not always possible or available (especially when used ever so occasionally). Trying to make do with free solutions, I tried quite a few of them, and frankly found many too frustrating or too convoluted to use. While some may argue that JtR is not an easy suite of tools to use, it really is after playing with it a bit. After experimenting extensively for a few weeks, I wanted to share my insights with the community.

Unlike previous articles this is a one-off, unless there is interest in looking at brute force heterogeneous JtR cluster-based cracking. We won't look at theory beyond what's needed to get it running as I suspect most of you are already familiar with the concepts.

Fortunately, I remembered that my passphrases were based on everyday English words, though modified (or mangled) somewhat. Mangling? Choosing JtR was a no-brainer (I could have used Hashcat but it got the better of me). Its wordlist mangling capability set the standard, copied by many others.

Assumptions

To follow along in this article it helps to know a bit about compiling software under UNIX/Linux, and how to navigate its command line. It also helps to have a basic understanding of Beowulf-like clustering or distributed processing. A basic knowledge of wordlists is also helpful. As always, try not to run your commands as root.

What is JtR?

JtR is an underrated password cracker supporting many modern and outdated hash algorithms. Easy to use, it has ample documentation (some errors here and there though), and is highly configurable, scaling beyond 512 CPUs and a few GPUs.

Unlike most commercial software (precompiled) solutions, in heterogeneous clusters it is best to recompile software for existing processor capabilities to maximize performance, which can provide a tremendous boost. By re-running the pre-compilation configuration script and recompiling, 2-4x increases were identified for more recent processor extensions (e.g., SSE4x, AVXx). Of course, successful compilation requires that library dependencies be resolved. JtR compiles under UNIX, Linux, BSD (including Mac OS X) and Windows (via Cygwin). It provides very modern capabilities, leveraging today's latest parallelization technologies, including MPI, OpenMP and OpenCL (CUDA support was dropped).

Despite claims to the contrary, heterogeneous JtR cluster deployments are not that difficult to build, which we will use to deploy JtR in its wordlist mode. Although fully capable of brute force cracking, known to JtR as incremental mode, we do not examine it in this article.

Why use it?

Password cracking is notoriously time-consuming; thus, every extra machine helps and leveraging Linux (or BSD) allows repurposing older systems. All you need is a network-deployed operating system, installed locally or booted over the network (use network booting for homogenous clusters).

JtR's "bleeding edge" and "jumbo" versions (available from Openwall.com and GitHub) are straightforward to use and surprisingly stable. I've used them in the past, running on a large heterogeneous cluster (those systems have since died).

Before taking out any other tool for wordlist and dictionary mangling, JtR should be your go-to tool. Some JtR GUIs exist but they do not let you fully capitalize on JtR in a clustered environment. Instead, use the command line; spending a few minutes reading JtR's documentation makes it intuitive to use.

While it helps to know some UNIX/Linux, JtR is no more complicated than an advanced DOS utility. However, if you are planning to run it under Windows, recompile it under Cygwin, and if it is to be used in a clustered environment, then Windows or Cygwin SSH needs to be installed and configured. Expect that the Windows version of JtR will have problems supporting OpenCL (importantly, it runs faster under CMD.EXE than from a Cygwin terminal). And if you can, stick with something *NIX-based.

Finally, JtR provides robust session recovery. Picking up from a saved state is straightforward and restores the session as they were.

What it's not

It's not ElcomSoft, Passware, or Hashcat (likely the fastest password cracker available today). Hashcat supports distributed computing via forks; some GitHub projects are devoted to simplifying this approach. ElcomSoft and Passware are very scalable but are expensive, prohibitively for some.

It depends on what you need or want to do. For sheer brute forcing, a very well respected colleague tells me Hashcat is definitely the fastest. But I'm old-fashioned, so I prefer JtR. However, it has some limitations, which we'll shortly look at.

High Points & Gotchas

After weeks of experimenting with JtR's salted SHA512 and EncFS CPU and OpenCL implementations (sha512crypt, sha512crypt-opencl, encfs and encfs-opencl), here is a list of what I found: ▷

Password cracking is notoriously time-consuming; thus, every extra machine helps and leveraging Linux (or BSD) allows repurposing older systems. All you need is a network-deployed operating system, installed locally or booted over the network.

Linux Thread Optimization

While JtR leverages very advanced programming concepts (OpenMP, OpenCL, MPI), each have their place. What we haven't discussed is how to further optimize the running threads. One thing that is rarely discussed in IT security is the concept of processor affinity. In short, the system scheduler, integrated into the core of the kernel (Windows & Mac are no exception here), oversees thread scheduling. In so doing, threads get moved around every so often, switching from core to core. That is the way it is, by default, on all modern systems. However, further thread optimization can be achieved by tweaking the scheduler–that is instructing it not to move threads around. As threads slow down, waiting for the next batch of data or instructions, depending on how long it takes, multiple threads can be migrated to a single core. All this thread switching adds overhead, which takes away from raw processing power. Since OpenMP is the way to go with respect to JtR in heterogeneous clusters, the primary environment variable to set for each JtR SSH instantiation session is OMP_PROC_BIND=TRUE. If you wish to keep threads isolated too specific cores (to partition up processing resources) then OMP_PLACES should be set. Additional information is available in the compiler's OpenMP documentation.

- For clusters, be consistent; keep all instances on the same local disk location or network share (e.g., /cracking/john). Keep logs, session and cracked password information here.
- MPI overhead is very high (could be the approach to MPI) so where possible, use OpenMP or --fork.
- Of the three task/thread distribution options --fork is the clear winner. OpenMP is a close second. And both are 3-4x better than MPI for local and remote threads.
- MPI can still be helpful, but deployment consideration is important.
- Clustering generates minimal network traffic.
- Do not allocate more than 1 CPU thread per GPU. Only one is required to keep one or more GPUs busy. Older versions may have been different. But since my systems were limited to one GPU I cannot say with certainty whether a 1-to-1 or 1-to-many approach is best.
- Session recovery for CPU and GPU sessions is straightforward and works with --fork, --node, OpenMP and MPI.
- It scales beyond 512 CPU cores with minimal network overhead - this was tested on a colleague's mid-size university cluster.
- Appropriate system cooling is a must.
- When the underlying network cannot be trusted for NFS or SMB use SSHFS.
- Once a session is underway, new systems cannot be added.
- When restoring a session, generally the same number of logical CPUs and physical GPUs previously used are required, although this can be bypassed.
- Allocate no more threads than the number of logical CPU cores and GPUs per system.
- Use OpenMP and MPI for homogenous clusters, with 1 MPI thread used to remotely instantiate with OpenMP running atop the remaining CPU cores.
- For heterogeneous clusters, use --node and OpenMP. To use this, it is necessary to configure a node-based performance-to-wordlist size (PTWS), which requires per system benchmarking.
- Set environment variable OMP_NUM_THREADS=X on a per host basis to allocate the number of OpenMP CPU cores to be used.
- For heterogeneous clusters, reconfigure and recompile for each system to ensure correct processor extensions are used where available.
- Use an MPI hosts file to correctly set remote thread distribution. MPI and OpenMPI hosts file configurations are slightly different.
- Compilation can be long; for multicore systems use make -j NUM to really speed things up.
- JtR provides some support for Intel's Xeon Phi coprocessor. Check out README-MIC file for more information.
- The default rules that come with current bleeding edge and jumbo editions are numerous, and in many instances sufficient for many wordlist cracking needs.
- JtR has the ability to run OpenCL on non-GPU devices (i.e., CPU). However, of all the systems I tested, most with NVidia GPUs, only my clustered laptop could list non-GPU OpenCL-ready devices (it used an embedded Intel HD graphics device). It seemed that there was some incompatibility between JtR and the Intel OpenCL libraries as whenever CPU or GPU OpenCL was used JtR would crash.
- In --node mode it's possible to add OpenMP threads to a recovered session; all that is required is rerunning JtR with its --session feature enabled with a modified OMP_NUM_THREADS. This would allow transferring one system's session to a larger CPU system.

2X XEON	I7 980X	I7 4500
1 core = 1324 c/s	1 core = 458 c/s	1 core = 1285 c/s
2 OMP = 2456 c/s	2 OMP = 909 c/s	2 OMP = 2272 c/s
4 OMP = 4632 c/s	4 OMP = 1830 c/s	4 OMP = 2217 c/s
8 OMP = 8444 c/s	6 OMP = 2616 c/s	N/A
16 OMP = 14637 c/s	8 OMP = 2508 c/s	N/A
32 OMP = 17018 c/s	12 OMP = 3010 c/s	N/A

Table 1. Number of OMP Threads to Benchmark

Experimental Setup

To conduct the wordlist attacks; I am using JtR bleeding jumbo obtained from GitHub (JtR magnumripper), dated November 10, 2017.

My cluster setup is heterogeneous, thus requiring a bit more work. The cluster is composed of three machines, a dual 8-core Xeon 2630v3 [32 logical cores (logical cores include actual physical cores and Hyper-Threading based threads. Thus, a HT-enabled processor yields two logical cores per physical core) with AVX2 support], and two i7 systems: an i7 980X (12 logical cores with SSE4.1 support) and an i7 4500 (4 logical cores with AVX2). For GPUs, I used an NVidia GeForce GT 720, an NVidia GeForce GTX 460 and an Intel HD Graphics, respectively (my GPUs are crappy). Only the Intel GPU was unstable and would crash during JtR OpenCL instantiation.

Building the Cluster & Compiling Jtr

I compiled JtR with OpenMP, MPI, OpenCL and experimental code support for each of the three systems. JtR's --fork capability is automatically compiled in.

All systems were configured with passwordless SSH (MPI uses it by default, and using Rsh just doesn't make sense, it's too insecure). No additional network configurations were necessary as the cluster was on its own physical network. All that was needed was a decent network switch (>100 Mbps), cables and configuring each system's /etc/hosts. Share and mount a common directory (e.g., /cracking/john) for storing all cracked passwords, hash files, JtR sessions and recovery logs.

As each clustered system is from a different x64 processor generation JtR must be "configured" and compiled for each system (thereby ensuring optimal performance), with all instances stored on the shared directory. However, before starting the cluster, each system must be benchmarked to determine how the wordlist cracking workload should be partitioned

CPU	OMP THREADS TO BENCHMARK
2x Xeon	1, 2, 4, 8, 16, 32
i7 980X	1, 2, 4, 8, 12
i7 4500	1, 2, 4

Table 2. Candidate Ciphers/Sec (c/s) Benchmark Results

Benchmarking

I prefer OpenMP handling JtR's multithreading because it is straightforward and easy to use. Importantly, testing and benchmarking (--test) is incompatible with --fork, another reason for leveraging OpenMP. There is talk about the usefulness of Hyper-Threading (HT), benchmarking will help us determine if HT helps or limits performance (the results are surprising).

I used the OMP_NUM_THREADS values set out in Table 1. We'll come back to this shortly.

From these results, we see that HT actually improves performance for the i7 980X and 2x Xeon, by 20% and 16.3%, respectively. However, for i7 4500 it actually decreases performance by 2.48%. I didn't believe the numbers so I reran the benchmarks 5 more times; Table 2 shows the averaged results.

I also benchmarked the GPUs using a single CPU thread:

```
$ ./john --test=120
--format=sha512crypt-opencl
```

This command benchmarks against all available GPUs in the local system. To list all GPU devices in a system use command:

```
$ ./john --list=opencl-devices
```

In a multi-GPU configuration, specify individual GPUs using --device=a,b,c... The NVidia GPUs results are found in Table 3. ▷

Password Cracking at Home

Supercomputing and password cracking at home: Password cracking used to be relegated to specific government departments, with large enough budgets to buy and run the supercomputers needed to achieve realistic results. Today, that is no longer true. Home users can match the performance of older government setups for a fraction of the price. With $10-20K, it is possible to achieve well over several hundred teraflops (1x1012 floating point operations/sec). Consider that the most powerful supercomputer today is Sunway TaihuLight, benchmarked at just over 93,000 Tflops. You don't need the latest and greatest GPU and certainly not the newest CPU to get bragging rights. Such systems are becoming mainstream among professional pentesters to crack enterprise passwords and encryption systems.

GPU DEVICE	PERFORMANCE
GeForce GT 720	2052 c/s
GeForce GTX 460	9395 c/s

Table 3. CPU Performance Metrics

For comparative purposes, the Hashcat equivalent of sha512crypt running on one GPU, obtained by a colleague, had the following results:

33.5K c/s on an NVidia GTX Titan
102.6K c/s on an AMD Radeon HD 7900

Wordlist Partitioning

Once the benchmarks are done, we need to know how to partition the wordlist. I am using the ROCKYOU wordlist, with just over 14.3 million entries. We need to divvy it up according to a system's processing power.

To do this we add up the c/s results from the CPU and GPU benchmarks results (Tables 2 and 3). Together, they yield approximately 33.7K c/s, which, when proportioned against each system's capabilities, can be used to generate a whole integer-based PTWS ratio, which we find in Table 4.

Recovering Shadow And TC Passphrases

To extract password hashes from a Linux system, we can use JtR's unshadow tool, which we run against the passwd and shadow files and pipe, to some file, similar to the following:

```
$ ./unshadow /mount/etc/passwd /
mount/etc/shadow > /cracking/
john/shadow.hashes.txt
```

With the PTWS ratio calculated, we can run JtR on each system according to a specific proportion, as shown in the following command:

```
$ ./john --node=1-170/337 --pot=/
cracking/john/cracked_passwords.
pot --save-memory=1 --wordlist=/
cracking/john/rockyou.txt
--rules=all --format=sha512crypt
session=/cracking/john/CLUSTER
_CPU_SET_X --progress-every=60
--verbosity=5 /cracking/john/
shadow.hashes.txt
```

john is JtR's password cracking program Most of the parameters are self-explanatory, except for a few described below.

CPU OR GPU	BASE RATIO (APPROX.)	PTWS RATIO
2x Xeon	17K c/s : 33.7K c/s	x10 = 170/337
i7 980X	3K c/s : 33.7K c/s	x10 = 30/337
i7 4500	2.3K c/s : 33.7K c/s	x10 = 23/337
GT 720	2K c/s : 33.7K c/s	x10 = 20/337
GTX 460	9.4K c/s : 33.7K c/s	x10 = 94/337

Table 4. PTWS Ratio Based Wordlist Partitioning

I have presented several ways of working with JtR in heterogeneous and homogenous cluster configurations while leveraging thread parallelization technologies, but the methods I selected are not the only ones.

The --node parameter is essential. Through it we tell john how to partition a specific wordlist. You provide a ratio, in whole integers, based on the ratio's start and end points, and john takes care of the rest. The numerators must add up to the denominator otherwise a part of the wordlist will not be processed. Then we rerun this command for all the other CPU and GPU nodes in the cluster according to their specific PTWS ratio. The parameter --pot specifies the file name and path of the cracked password output file. Verbosity (from 1 to 5) is important for status reporting, but not so much that it overwhelms the console or consumes significant bandwidth. Providing a progress update every 60secs in often sufficient.

The file created by -session (and associated files) is used for session recovery. We substitute the X in CLUSTER _CPU_SET_X for 1, 2, and 3 for the systems in the cluster. To keep things simple, differentiate the session name between CPUs and GPUs.

For GPU JtR computing, we need to provide GPU-specific information, specifically the GPU device to use (parameter --device) and we need to specify an OpenCL based hash format. The following command will help:

```
$ ./john --node=X-Y/Z --pot=…
--save-memory=1 --wordlist=…
--rules=… --format=sha512crypt-
opencl --device=0 --session=…
GPU_X --progress-every=…
--verbosity=5 hashfile
```

For systems with more than one GPU, we set --device=1 if we only want to use the second GPU; if we want two GPUs, we set it to --device=0,1 or to "gpu".

Everything we have done is fully scriptable and automatable, which is highly recommended for larger clusters. It took about five days to find the password, which was then used to unlock the TC volume.

Notes For Homogenous Clusters

A homogenous cluster requires less work, you can instantiate all CPU workload distribution using a single MPI command. GPU distribution requires a second command, with hash format and device to suit. With a correctly defined MPI hosts file, the following command can be used, executed just once:

```
$ mpiexec --hostfile mpi-nodes.txt
./john --pot=… --save-memory=1
--wordlist=… --format=… --rules=…
--session=… --progress-every=…
--verbosity=… hashfile
```

EncFS

EncFS is a FUSE-based encrypted disk file system. The password I had used only a few weeks prior had completely escaped me. But I remembered it was based on items from my personal and professional life. Thus building up as complete a wordlist as I could consisting of persons, pets, events, places I've been, things I've seen, and common password patterns and symbols I use.

I tried many wordlist generation tools, including Hashcat Utilities' combinator. It was an interesting program but did not fulfill my requirements. In the end, I settled on the Ruby program RSMangler. To generate a complete wordlist based on input words, run as follows:

```
$./rsmangler.rb --file input_
words.list -d -t -T -c -l -u -e
-i --pna --pnb --na --nb -y -C -a
--allow-duplicates --punctuation
-x 40 -m 20 --output generated-
wordlist.txt
```

As an interpreted language, Ruby is very slow. After more than 36 hours, I stopped the program. My output file had over 470 million lines, and with a benchmarked capability of just over 1K c/s, it was going to take forever. To speed things up, I sorted the wordlist file, using sort, and then grepped for specific starting sequences, which I knew I always use. After more processing, I succeeded in bringing down my wordlist candidates down to around 8 million, something that could be done in weeks. Of course, not all mangling rules would have been carried out, but at this point I didn't really care, as I was not particularly optimistic about getting the passphrase back.

I used JtR tool encfs2john.py to obtain the hash. Using the same cluster, CPUs and CPUs, I ran amended versions of the previous commands. I had to benchmark the systems against hashes encfs and encfs-opencl to find my PTWS ratios.

It took 22 days to crack the hash.

With the password in hand I was then able to mount the EncFs file system and regain access to my files, which I thought were lost.

Conclusion

JtR is a highly capable wordlist mangling password cracker. I have not thoroughly tested its brute force capabilities, but it can hold its own. Life would have been simpler with a homogenous cluster, but sometimes we have to make do. In the end, I was approaching around 31K for sha512crypt and and 1K c/s for encfs. Keep in mind that my GPUs were not very powerful. True overall throughput was closer to around 22 c/s and 750 c/s, respectively. Comparatively speaking, Hashcat is faster than JtR, but not by magnitudes.

At this time, neither Hashcat, ElcomSoft nor Passware support EncFS so comparisons were unavailable for that hash.

I have presented several ways of working with JtR in heterogeneous and homogenous cluster configurations while leveraging thread parallelization technologies, but the methods I selected are not the only ones. Other thread cluster-based parallelization techniques exist, but I believe the ones I used are the most relevant ones on the hardware I had available. If I had access to multi-GPU systems, I would have evaluated the best thread-to-GPU ratio but this was not possible. If anyone out there finds this number, please let the community know by putting it up on Openwall.com or the JtR magnumripper GitHub (as contributed documentation). •

Making the Most of What You've Got

You don't need the latest and greatest hardware and software to start exploring the world of password cracking, just follow these guidelines:

- *Never run password cracking or any related commands as an administrative user.*
- *Compile JtR under Cygwin using GCC; you'll need to make certain all the necessary libraries are installed - I recommend a full Cygwin installation to avoid library dependency torture.*
- *Don't run JtR under Cygwin's bash, it's slower than running it under CMD.EXE.*
- *You will need an SSH server and client; many pre-compiled solutions are available for this. You can also configure Cygwin to instantiate an SSH connection but this is more work than its worth for a program you will be running from CMD.EXE.*
- *OpenCL is likely to be buggy, but mileage will vary.*
- *Use SMB to share files between Windows and Linux (or other) systems; it's straightforward to configure.*
- *Ensure security and policies are retained.*
- *Make sure management is OK with password cracking; sometimes it doesn't sit well with them.*

Richard Carbone *has been working for Defence R&D Canada since 2001. He has been working as an infosec researcher for the last eight years and has published numerous articles and case studies. He is also an open source software expert and was the co-author for a Government of Canada study that influenced federal government policy on the adoption of open source. Finally, he is a certified digital forensic investigator, incident handler and malware reverse engineer. He can be reached at val-forensics@drdc-rddc.gc.ca.*

DOCUMENT CONTROL DATA

*Security markings for the title, authors, abstract and keywords must be entered when the document is sensitive

1. ORIGINATOR (Name and address of the organization preparing the document. A DRDC Centre sponsoring a contractor's report, or tasking agency, is entered in Section 8.)

Digital Forensics Magazine

2a. SECURITY MARKING (Overall security marking of the document including special supplemental markings if applicable.)

CAN UNCLASSIFIED

2b. CONTROLLED GOODS

NON-CONTROLLED GOODS
DMC A

3. TITLE (The document title and sub-title as indicated on the title page.)

Wordlist password cracking using John the Ripper: A tutorial and lessons learned for heterogeneous clusters

4. AUTHORS (last name, followed by initials – ranks, titles, etc., not to be used)

Carbone, R.

5. DATE OF PUBLICATION (Month and year of publication of document.)	6a. NO. OF PAGES (Total pages, including Annexes, excluding DCD, covering and verso pages.)	6b. NO. OF REFS (Total references cited.)
February 2018	6	0

7. DOCUMENT CATEGORY (e.g., Scientific Report, Contract Report, Scientific Letter.)

External Literature (P)

8. SPONSORING CENTRE (The name and address of the department project office or laboratory sponsoring the research and development.)

DRDC – Valcartier Research Centre
Defence Research and Development Canada
2459 route de la Bravoure
Quebec (Quebec) G3J 1X5
Canada

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.)	9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)
05AA PASS	

10a. DRDC PUBLICATION NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.)	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)
DRDC-RDDC-2018-P025	

11a. FUTURE DISTRIBUTION WITHIN CANADA (Approval for further dissemination of the document. Security classification must also be considered.)

Public release

11b. FUTURE DISTRIBUTION OUTSIDE CANADA (Approval for further dissemination of the document. Security classification must also be considered.)

12. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Use semi-colon as a delimiter.)
password cracking; wordlists; John the Ripper; sha512; encfs
13. ABSTRACT/RESUME (When available in the document, the French version of the abstract must be included here.)